Development of a payment channel over the Bitcoin network

David Lozano Jarque, *Undergraduate student, UAB.cat*

Abstract—Bitcoin is a decentralized digital cryptocurrency that allows payments between users without the need of a central authority. Despite the potential of the technology, in the past years, the scaling debate has been the main focus of development as because of the internal details of implementation of the technology, the network can not process and store the highly increasing demand of transactions in the public ledger, also called the *blockchain*. A solution for this is reducing the need of transactions with off-chain payment channels, that can be able to process thousands of micropayment transactions between two nodes so that most transactions do not appear in the blockchain but if they did would be valid, using the Bitcoin scripting language and some game theory techniques. With payment channels, only the setup and closure transactions would appear in the blockchain and all the payment transactions would be temporary and stored just by the nodes of the channel, relieving the Bitcoin blockchain transaction rate. This project consists in designing and implementing a bidirectional payment channel by using the combination of two unidirectional payment channels.

Keywords—Cryptocurrency, Bitcoin, scaling, Payment channel, Bidirectional payment channel

1 Introduction

BITCOIN is a cryptocurrency that first appeared in a cryptography mailing list [1] with a post by an anonymous user who called himself "Satoshi Nakamoto" and defined in a whitepaper [2] the first decentralized cryptocurrency. It allowed direct peer to peer digital currency transactions without the need of a central authority in which users trust for validating those transactions. Instead, each peer can validate those transactions using cryptography (technically validating digital signatures) and after that generate a block of transactions including them. An action (validate transactions and generating blocks) whose reward is retrieving newly generated currency, aiming peers to secure the network. To decide which node can generate (also called *mine*) the next block of transactions and reach a consensus, they are challenged to solve a cryptographic riddle, called *proof of work* [3]. Nodes trying to solve that challenge and generate (also called *solve*) new blocks to receive a reward for their work are called *miners*

All the transactions ever made, grouped in a structure called *block*, are stored forming a chain. This chain is stored in a distributed read-write only database each (full) network node stores called the *blockchain*. This name is given as each block is chained to the previous creating a not-modifiable chain of blocks using hash functions that link each block to a previous one using its hash.

1.1 The *blockchain* limits

At a high level, this is how Bitcoin works. The problem comes with the public ledger or *blockchain* that stores absolutely all transactions ever performed. With an average block size of nearly 1MB [4] (as it is the hardcoded limit size for a block) that contains approximately 2.000 transactions [5] and with a block appearing every 10 minutes, this makes this distributed database grow approximately 50GB every year [6]. The block size limit is fixed at 1MB and difficulty for solving new blocks using the proof-of-work algorithm [3] is dynamically adjusted so that new blocks appear approximately every 10 minutes. This

- *E-mail: uab@davidlj95.com*
- *Specialized in Information Technologies*
- *Tutored by Joan Herrera Joancomart (dEIC.UAB.cat)*
- *Course 2016-2017*

Manuscript written on June 2017, Engineering School (UAB.cat)

is fixed in the protocol and therefore the code of the software nodes run, so can not be changed without everyone agreeing or could lead to a chain split [7].

1.2 The scaling problem

With Bitcoin gaining popularity among more users, more transactions are created and needed to handle and get stored in the blockchain. Due to the limits set, not all transactions can be handled and the number of delayed transactions until the network can handle them is increasing every day. There are several active proposals [8], [9] to change those limits and allow to handle more transactions, but meanwhile a solution gets activated and agreed by all the Bitcoin ecosystem (users, developers and miners), another long term solution is being proposed: reducing the number of transactions needed to perform payments.

1.3 Payment channels

This is where payment channels appear [10], allowing to two users or more that need a constant flow of transactions to pay each other instantly without waiting for the confirmation of the transaction in the blockchain. The way they operate is exchanging transactions privately between them that do not appear in the blockchain, also called *off-chain transactions*. Just the opening and closure transactions of the channel would be needed to appear in the blockchain, therefore reducing the amount of transactions they need to send to the blockchain and relieving the blockchain from transactions. The opening would lock some funds into a smart contract and the closure would return those funds depending on the transactions performed in the payment channel.

In the event of any dispute on the closure transaction, the trick is that privately exchanged off-chain transactions could be sent to the blockchain and they would be valid. Therefore once broadcasted would give the same result of funds distribution as the closure transaction. Those payment transactions, but, are kept private by each node until the channel needs to be closed and there is no mutual agreement between peers. Each payment transaction replaces the old one using incentives so just the last one payment needs to be kept, allowing a high rate of transactions between nodes of the payment channel. The payment channel has to be secure by design and implemented with a secure protocol so that no party of the payment channel can not steal or lock the other party funds (act maliciously).

2 Bitcoin and Smart Contracts

As said before, Bitcoin allows to store a decentralized consensual database of transactions that transfer units of currency (bitcoins [11]) between users. To understand how currency units are moved, we need to understand what a transaction is at a low level technical detail

2.1 Bitcoin transactions

A Bitcoin transaction is just an array of bytes that specifies some inputs and some outputs, prefixed by a version field and suffixed with a field named *nLocktime* we will talk about later. What every transaction does is to spend

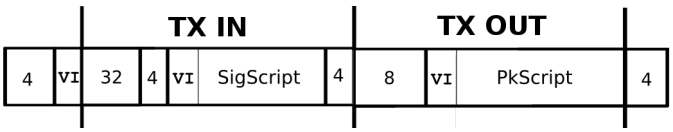

Fig. 1: Transaction binary format

a previously generated output by specifying in an input a pointer to that previous output, also called *UTXO* (unspent transaction output). An *UTXO* refers to a transaction id and number of output of that transaction that has not yet been spent by any other transaction[1]. Also, some

1. If all transactions have to spend a previous output, when are the first outputs generated? There is a special transaction with no inputs that is the one that generates currency units. It is just valid once in a block, to spend the reward (that must match exactly the reward value) a miner receives when solving a new block

data (most times ECDSA signatures) follows the *UTXO* in order to authorize spending the funds. This data is called the *scriptSig*. This input or inputs are moved specifying a new set of outputs to move the funds spent. In each output, the value of currency units to move to the output and the conditions for them to be spent must be specified. Those conditions are placed in a field called *scriptPubKey* as initially funds were always paid to a public key, so just the owner of its pairing private key could spend them.

This mentioned data to spend *scriptSig* and conditions to spend *scriptPubKey* is specified using a scripting language exclusively designed for the Bitcoin protocol [12].

2.2 Bitcoin scripting language

One of the powers of Bitcoin is its stack-based scripting language, as allows to specify how funds can be transferred by creating scripts in the Bitcoin scripting language [12]. Therefore for a transaction to be valid, the input must refer a valid and non-spent *UTXO* and the execution of the *scriptSig* input followed by the referred output script (the *scriptPubKey*) must end successfully with a non-empty stack. Also, the sum of outputs' values must be less than the sum of inputs' values[2]. This scripting language basically reads 1-byte opcodes that able to store (push into the stack) data, perform arithmetical and logical operations, and some cryptographic operations like ECDSA signatures and hash functions among others.

The most used script set to move funds is called a P2PKH (pay-to-public-key-hash). This kind of script set uses the following scripts to move funds:

- ***scriptPubKey*** Specifies a hash of an ECDSA public key and a signature from this public key whose hash matches the specified one
- ***scriptSig*** Must contain a valid signature followed by the public key used to create that signature (whose hash must match the specified in the *scriptPubKey*)

A Bitcoin address is then the hash of a public key needed in the mentioned *scriptPubKey*[3]. This addresses are commonly used to pay units of currency between users who reveal their addresses to be paid.

But as said before, the Bitcoin scripting language allows us to code any script to specify the spend conditions or *scriptPubKey* and any script to specify the data needed to spend following those conditions (the *scriptSig*). Here is when the script set called P2SH (pay-to-script-hash) comes. This method of payment allows us to create an smart contract by defining an script where we specify the conditions to spend the output (called the *redeemScript*) and create an output paying to this script hash:

- ***scriptPubKey*** Specifies the hash of the smart contract (defined in a *redeemScript*) that must be executed to spend the funds
- ***scriptSig*** Contains the data needed by the *redeemScript* in order for it to execute succesfully along with the *redeemScript* itself.

As we see to spend a P2SH UTXO, we must reveal the *reedemScript* and often specify also data that the script needs to be spent, like some signatures (multisig P2SH) or a hash preimage, or whatever the *redeemScript* we design needs to execute succesfully[4].

3 Unidirectional Payment Channels

Once understood how Bitcoin transactions work and how we can develop smart contracts

2. The difference between the sum of inputs' values and the sum of outputs' values if is greater than 0 is called the transaction **fee**, and will be rewarded along with the block reward to the node that includes the transaction in a block

3. Technically, the address is prefixed by a version byte and suffixed with a SHA-256 4-byte checksum of that hash, all encoded in base58 for visualization purposes. The version byte helps identifying the address script set (P2PKH or P2SH) and network

4. Despite we could technically specify any output and input script so that if the input script followed by the output script execute succesfully the transaction is valid, if we don't use either P2PKH or P2SH, our transaction would be non-standard and probably not accepted by the network nodes because of the Bitcoin protocol implementation [13]

using the Bitcoin scripting language, we can introduce unidirectional payment channels. This kind of payment channel basically ables transactions between two users, one of them paying (payer) incrementally some amounts to the other one (payee). We will call Alice the payer and Bob the payee.

Also called simple micropayment channels, they were first defined by Mike Hearn and Jeremy Spilman [14]. With the activation of CLTV opcode in the Bitcoin scripting language through BIP-65 [15], but, those channels were improved to avoid transaction malleability [16] simplifying the channel structure.

3.1 The scheme

Every channel has three phases:

1) **Funding:** where Alice, also called funder, puts some units of currency she owns into a smart contract (we use a P2SH to pay to a *redeemScript* hash). The transaction to perform this operation is called the *funding transaction*. This smart contract must lock the funds for a certain period of time in order to avoid Alice to spend the channel funds before the channel gets closed. The time where the funds get unlocked and available to Alice again is called the expiry time of the channel. This way we ensure Alice can not move this funds until the channel's expiry time so Bob can retrieve the payments before this time with any of the payment transactions signed by both of them.
2) **Payments:** where Alice creates and signs transactions spending the funding transaction UTXO that incrementally pay more to Bob (via a P2PKH *scriptPubKey*). Bob just keeps the transaction that pays more to him, as just one of all the payment transactions is valid because all of them spend the same UTXO (and just one transaction can spend an UTXO). This is why the channel is unidirectional, as Bob will keep the transaction that pays more to them because of the economical incentive. All these transactions are not released to the blockchain until the channel closure, where Bob performs his signature if he agreeds in the transaction output (moves the funds to his P2PKH address, for instance) and releases the transaction to the network. A multisignature scheme (also called *multisig*) [17] is necessary to ensure Bob can not perform any payment by himself and Alice can not return the funds to herself. **Closure:** this can happen because of two reasons:
    - **Graceful close** Bob broadcasts the latest received payment transaction (signed by both Alice and Bob) to spend the funding transaction UTXO and closes the channel as the funding transaction UTXO can not be spent again. This must be performed by Bob before the channel's expiry time in order to ensure that funds can be retrieved as after that time Alice can move the funds to herself again.
    - **Expiry date** If Bob does not cooperate, when the expiry time arrives Alice can safely recover her funds just by performing a P2PKH transaction spending the funding transaction UTXO.

To sum up, the scheme is to create a funding transaction paying a certain amount of locked Alice's funds to a smart contract that allows spending it either

- a) Using a *multisig* scheme so Alice creates a transaction to pay to Bob that signs prior sending it to him. When Bob has the partially signed transaction, if he agrees paying to the output specified (probably his P2PKH address) he can just sign it and wait to broadcast it before the expiry time and make the payment effective
- b) After a certain time by Alice (as if Bob does not collaborate and does not perform the *multisig*, funds could be locked forever)

This can be achieved either by creating a smart funding transaction that includes the expiry time condition or a *multisig* funding transaction and a refund transaction that is signed by both

and allows to be spent after a certain time using the mentioned CLTV opcode defined in BIP-65 [15]. For this project, we opted for a single smart transaction in order to simplify the process and avoid transaction malleability. Eventually, the most important part is the funding smart contract, as must allow a refund after a certain time in case Bob does not collaborate so Alice can recover the funds and also to pay incremental amounts to Bob.

3.2 The smart contract

In order to create a transaction that spends some funds of Alice and the output pays to a smart contract that requires a *multisig* for being spent or just a signature after certain time, our proposal[5] was to create a transaction funding this *redeemScript*:

```
OP_IF <time> OP_CHECKLOCKTIMEVERIFY
OP_DROP <PubKeyAlice_1> OP_CHECKSIG
    OP_ELSE OP_2 <PubKeyAlice_2>
 <PubKeyBob> OP_2 OP_CHECKMULTISIG
             OP_ENDIF
```

Note that Alice both owns private key of `<PubKeyAlice_(1/2)>` and Bob holds the private key of `<PubKeyBob>`.

3.3 Channel operations

With this smart contract script, we could create and test after that all the transactions for the channel:

- **Funding**: A transaction spending an input referring to an Alice's P2PKH UTXO and with a P2SH output paying to the previously mentioned redeem script hash
- **Payment**: A transaction signed by both parties (firstly signed by Alice and then sent to Bob missing its signature to be valid) spending the redeem script with the `OP_CHECKMULTISIG` statement specifying an `OP_FALSE` and whose outputs are two P2PKH to Bob for some amount and to Alice as a return. Each payment transaction must pay more than the previous one to Bob, as Bob will always hold the one that pays more to him. In case of wanting Bob to receive less than the previous transaction, we need a bidirectional payment channel.
- **Graceful closure**: A payment transaction can act as a closure if broadcasted to the network previously signed by Bob. It has to be sent by Bob user before the expiry time or Alice could use the *refund transaction* so all payment transactions would be invalid as those funds would be already spent by the *refund transaction*.
- **Closure by expiry time**: Also called *refund transaction*. A transaction signed by Alice, and with `nLocktime`[6] field set after the `<time>` field specified in the script. In other words, after the channel expiry time. This way Alice is spending the funding transaction with just its signature as specifies to pay with the first block of the redeem script with an `OP_TRUE` whose output is a P2PKH output to a public key she owns its associated private key.

3.4 The protocol

All this transactions must be created following a secure protocol that ensures all users are secure creating and operating the channel without any of them trusting the other. The protocol to establish a unidirectional payment channel between Alice (the payer) and Bob (the payee) would be the following: Alice, as the payer, requests opening a channel and specifies the funds of the channel (maximum amount Alice can pay to Bob) and the expiry date of it. If Bob agrees on the channel creation, he sends its public key so that Alice can create the funding transaction. Once the funding transaction is created, Alice sends the transaction to Bob along with the redeem script, so he can trace it and verify the contract is correct. Bob sends an acknowledge to Alice if wants to proceed with the channel opening (a signed one with Bob's key so Alice can verify the acknowledge is real). Alice eventually can broadcast the funding transaction. Once the

5. Along with Carlos Gonzalez Cebrecos

6. We require the use of the `nLocktime` transaction field in order to make the script `OP_CHECKLOCKTIMEVERIFY` work as specified in BIP-65 [15]

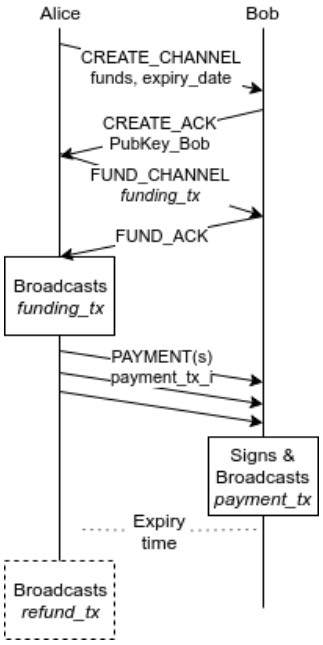

Fig. 2: Unidirectional payment channel protocol

transaction is confirmed, Alice can create payment transactions, sign them and send them to Bob privately. When the expiry date is arriving, Bob will close the channel by broadcasting the latest received payment transaction. If Bob does not collaborate, Alice can create and broadcast her refund transaction after the channel expiry time.

4 BIDIRECTIONAL PAYMENT CHANNELS

The problem with above channels is that just Alice can pay incrementally amounts of currency unit to Bob. What if we want the channel to be duplex so that both parties can send amounts of currency in both ways? In this work we researched following the solution proposed by Christian Decker and Roger Wattenhofer [18] that is to basically to create a duplex payment channel by using two unidirectional payment channels linked together, one in each direction. Another popular solution proposed is the Lightning Network, that uses a more complex structure to build a duplex payment channel based on hash-based smart contracts [19]

4.1 The scheme

As said previously, the idea is to use two unidirectional payment channels, one in each direction, so that we can pay in both directions. To do that, in the funding transaction, there must be two (or more) inputs and two outputs. One (or more) input and one output per user. The Alice input spent value minus fees will be the first output value, where the output will pay to the same redeem script as the unidirectional channel. This will be the channel used by Alice to pay to Bob. The second (or after last Alice's) input and second output will be constructed using the same scheme for Bob to pay Alice. The rest of the payment channel would work the same way that in a unidirectional channel, where each transaction spends an output or another depending if Alice is paying to Bob or viceversa.

4.2 The protocol

In order to create the duplex payment channel, the following protocol must be followed in order for the channel to be secure: We can

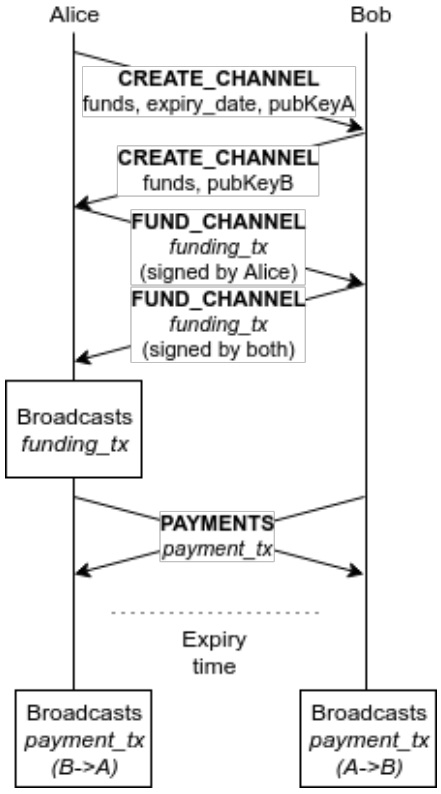

Fig. 3: Bidirectional payment channel protocol

see the protocol is similar to the unidirectional payment channel. In this example, Alice starts the request for the creation of the payment channel, but Bob could also send the request, inverting the communications until the payments section. The basic protocol consists in Alice sending the request to Bob for the channel creation, as happened with the unidirectional channel with the funds Alice desires to pay Bob, but also including his public key so that Bob can verify the funding transaction created. If Bob agrees with the channel creation, replies with his public key to create the output for Alice paying to Bob and the funds that Bob wants to use to fund his channel to pay Alice. Once Alice has all data, can create the funding transaction with the two outputs and her input(s), and sign her input(s)[7]. Alice sends the partially completed transaction (along with the redeem scripts) to Bob. Bob checks the transaction is correct and adds his input signed, returning the fully signed transaction to Alice as a final acknowledge for creating the channel. Once Alice receives the transaction, checks that is valid and broadcasts the transaction to the network. Now payments spending the Alice output to pay to Bob and the Bob output to pay to Alice can be performed creating off-chain payment transactions the same way as in unidirectional channels. To close the channel, both Bob and Alice have to release the latest received payment transactions before the channel expiry to close the channel. If a party does not collaborate, they can both send their respective refund transactions.

4.3 Channel operations

The same operations applied for the unidirectional payment channel (funding, payment, graceful closure and closure by expiry time) would be valid (despite the transaction for funding being slightly different with an added input and output for the second way channel).

4.4 Channel reset

One thing that can happen is that either Alice or Bob spends all the funds they owned paying to the other user. In that case, the channel needs to be reset, so that the received funds from the other party can be used to continue paying to them. To do this, a solution is also described by C. Decker and R. Wattenhofer [18] and is called the invalidation tree using what it is called atomic multiparty opt-in transactions.

4.4.1 Atomic multiparty opt-in

This kind of meta-transactions are a model for creating transactions to fund smart contracts (one or more outputs) that instead of being funded by one or more inputs with a P2PKH *scriptSig* owned by a user, they claim a multisig *P2SH* output that has not been signed yet. This allows to first design the smart contract and once all parties agree, they sign a transaction spending one or more P2PKH to fund the multisig output claimed by the opt-in transaction and now both transactions have funded the smart contract in a secure way no matter the order of signatures.

4.4.2 Locktime incentives

This previous transaction models are not necessary for a simple duplex payment channel, but can be used if we wish the channel to be reset. Creating another smart contract with different conditions (like specifying different amounts) spending the opt-in transaction but with a lower locktime (locking the transaction to be valid to a time closer to the present) than the previous smart contract transaction would make the new transaction the valid one per incentive as the old one would have a larger locktime and therefore the current one can be spent before. In order for the locktime incentive to invalidate previous transactions work, it must be lower than the channel's expiry time. Consequently, renewing the expiry time is the only channel parameter change that could not be done with this kind of incentive. We can also chain opt-in transactions forming what is called an invalidation tree, where the invalidation is performed by specifying lower timelocks on each new transactions branches to invalidate previous ones.

7. indicating `SIGHASH_ALL` meaning that signs the transaction containing the two outputs

4.4.3 Invalidation trees

Chaining multiple atomic multiparty opt-in transactions forming a tree can be used along specifying increasingly lower locktimes in order to invalidate old tree branches as those branches of transactions having a larger locktime would be replaced by ones with lower locktime because they can be released to the network earlier. All locktimes in order to safely invalidate old branches, they must be lower than the previous higher locktime by an increment of time enough for the transaction be confirmed on the network to avoid attacks (for instance, 3 or 4 blocks of increment). Also, these trees slightly change the closure by expiry time operation, as now, refund transactions are needed because we can not set a smart contract with two output values in the same output. This refund transactions would be created in the protocol along with the funding's channel operations. In the case of a graceful closure, a transaction spending the funding transaction with the final balance could be sent if both parties agree, or all the latest valid tree branch if they do not agree.

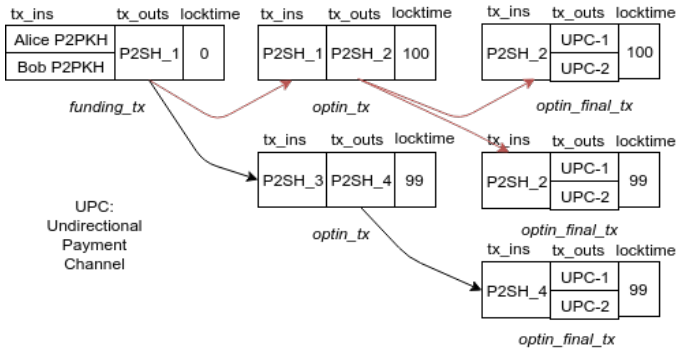

Fig. 4: An example of invalidation trees using atomic multiparty opt-in transactions and locktime incentives

5 The Implementation

In order to implement the bidirectional payment channel, a research was performed to check what Python libraries where available to develop smart contracts and therefore transactions with smart contracts. What we found is that no object-oriented and well documented library was available to create non-typical transactions (P2SH with a custom `redeemScript`). Because of that, we implemented a new library / framework to create easily customized transactions using the Bitcoin protocol information and it's implementation details [20], [21].

5.1 Our Bitcoin framework

To implement our framework, we decided to create a series of modules and classes oriented towards to the puzzle-friendlyness property: all objects / classes must be able to be serialized / deserialize into / from an array of bytes compatible with the Bitcoin protocol. We just implemented to save time, but, the strictly necessary modules and classes needed for this project development. With this framework we also hope to set the base to develop a well designed, usable and easy to understand Bitcoin Python library that aims new developers to create smart contracts in the Bitcoin network.

5.2 Developing progress

The framework started with the ability to create an empty valid transaction and after that implementing all the fields necessary, composing each field of another subfields to allow the mentioned puzzle-friendliness property. The latest developed part of the framework was part of the Bitcoin scripting language (that is in constant development) to implement the needed opcodes to create the smart contracts.

Once the framework allowed to create valid transactions (that required special focus on cryptography functions and its serialization), we tested basic P2PKH transactions created with the framework and a P2SH *multisig* transaction. After that, the `OP_CHECKLOCKTIMEVERIFY` opcode was implemented and tested and a unidirectional payment channel was created.

After all development and testing finished for the creation of valid and functional unidirectional payment channels, I started developing the bidirectional payment channel

as specified in the previous chapter of this document.

5.3 The duplex channel implementation

The channel implemented basically allows using a command line interface to operate a channel with the following operations. No communication has been implemented to focus on the channel security rather than automatization and easing the channel use.

- **Funding:** generates the funding transaction and an invalidation tree of transactions with a defined depth, accepting parameters to set the channel funds, expiry time and the public keys of the invalidation tree's P2SH scripts. In order for the invalidation to be secure, the P2SH scripts hashes have to be different so they can not match another node of the tree. For this purpose, we added in the implementation numbers at the end of the *redeemScript* that modify their hash but not their functionality. Once the funding transaction and the first invalidation tree branch has been created, the refund transaction is also created with the timelock set at the expiry time. All this transactions (except funding) will be signed by the user who creates the channel, that is supposed to also have the details of the channel as the software does not implement external communications. After that, they can be sent to the other user, who can use the `bitcoin-cli` utility from Bitcoin Core [**?**] to sign all of them. Then, the other user can return them to the creator so can eventually sign and broadcast the funding transaction.
- **Payment:** With the previous transactions stored, with the payment operation, setting the payment channel UTXO and with the private key, both users can generate payment transactions until the unidirectional payment channel of each of them is spent.
- **Reset:** Given the same parameters as the funding, but specifying a reset operation and the previous timelock used, will generate another branch of transactions with the new funds provided.

5.3.1 Usage

The script syntax is the following:

```
python -m src <operation>
[arguments]
```

Where we can use the optional argument `-h` to know how to indicate the operation (currently `fund`, `reset`) and rest of arguments

5.3.2 Future work and research lines

Usability

The current channel requires the users that operate the channel high knowledge about the Bitcoin technology and protocol as have to sign manually some transactions and broadcast them. In order to make this bidirectional payment channels accessible to a wider audience, the software should be automated to perform all operations with a graphical user interface. This interface should also hide the channel complexities: storing the transactions' tree, communicating both users to agree on the channel parameters, broadcasting transactions to the network and handling private and public keys mainly.

Multihop payment channels

Using HLTC (Hash-locked timed contracts [22]) the implementation could be extended providing off-chain transactions to perform payments across multiple existing payment channels in a similar way the Lightning Network implements it [19]

6 Conclusion

Bitcoin has a great potential as it's the first decentralized cryptocurrency currently understood as a great store of economic value. Despite that, the scaling problem makes Bitcoin growth slowlier than desired. A solution for that is relieving the Bitcoin's blockchain from transactions using payment channels with *off-chain transactions* between payment services providers [23]. The bidirectional payment channel described and implemented in this project allows to create simple and secure[8] bidirec-

8. Until SegWit [8] is not activated, just unidirectional payment channels and bidirectional payment channels without the reset operation are secure, because creating off-chain transaction chains can be vulnerable due to transactions' malleability issues [16]

tional payment channels using the combination of unidirectional payment channels, atomic multiparty opt-in transactions and invalidation trees with locktime incentives. Despite the channel's decreasing duration because of the reset operations, the structure is far more simple than other solutions like the Lightning Network [19] and requires less data exchange. Furthermore, if both parties cooperate along the channel creation, the bidirectional payment channel gets really simple to operate, with the disadvantage of having to send all the valid tree branch if the final balance of the channel is not mutually agreed. If the current project gets implemented securely with SegWit [8] activation, eventually Bitcoin will be able to provide a high throughput of instant and low fee transactions without worrying about its scalability.

ACKNOWLEDGMENTS

This work could not have been possible without the collaboration with Carlos Gonzalez Cebrecos, another student performing a similar project that coworked developing our Bitcoin framework and solving some Bitcoin and related technologies doubts along with our tutors Jordi Herrera Joancomarti, Sergi Delgado and Cristina Perez who introduced us in the Bitcoin world.

REFERENCES

[1] S. Nakamoto, "Bitcoin p2p e-cash paper." http://www.mail-archive.com/cryptography@metzdowd.com/msg09959.html, November 2008. (Accessed on 06/09/2017).

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, November 2008. (Accessed on 06/10/2017).

[3] "Proof of work - bitcoin wiki." https://en.bitcoin.it/wiki/Proof_of_work, November 2011. (Accessed on 06/10/2017).

[4] B. L. S.A.R.L., "Average block size - blockchain." https://blockchain.info/es/charts/avg-block-size. (Accessed on 06/10/2017).

[5] B. L. S.A.R.L., "Average number of transactions per block - blockchain." https://blockchain.info/es/charts/n-transactions-per-block. (Accessed on 06/10/2017).

[6] B. L. S.A.R.L., "Blockchain size - blockchain." https://blockchain.info/charts/blocks-size. (Accessed on 06/10/2017).

[7] A. Pace, "Guest post: Chain splits and resolutions bitcoin magazine." https://bitcoinmagazine.com/articles/guest-post-chain-splits-and-resolutions/, March 2017. (Accessed on 06/24/2017).

[8] "Segwit resources." https://segwit.org/. (Accessed on 06/10/2017).

[9] "Bitcoin unlimited." https://www.bitcoinunlimited.info/. (Accessed on 06/10/2017).

[10] "Payment channels - bitcoin wiki." https://en.bitcoin.it/wiki/Payment_channels. (Accessed on 06/10/2017).

[11] "Correct use of the word bitcoin - bitcoin stack exchange." https://bitcoin.stackexchange.com/questions/20901/correct-use-of-the-word-bitcoin, January 2014. (Accessed on 06/24/2017).

[12] "Script - bitcoin wiki." https://en.bitcoin.it/wiki/Script. (Accessed on 06/10/2017).

[13] "What node implementations and mining pools relay and process 'non-standard' scripts? - bitcoin stack exchange." https://bitcoin.stackexchange.com/questions/23435/what-node-implementations-and-mining-pools-relay-and-process-
March 2014. (Accessed on 06/24/2017).

[14] M. Hearn and J. Spilman, "Contract - bitcoin wiki." https://en.bitcoin.it/wiki/Contract. (Accessed on 06/11/2017).

[15] P. Todd, "bips/bip-0065.mediawiki at master bitcoin/bips." https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki, October 2014. (Accessed on 06/11/2017).

[16] "Transaction malleability - bitcoin wiki." https://en.bitcoin.it/wiki/Transaction_Malleability, August 2015. (Accessed on 06/25/2017).

[17] "Multisignature - bitcoin wiki." https://en.bitcoin.it/wiki/Multisignature, January 2017. (Accessed on 06/25/2017).

[18] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*, pp. 3–18, Springer, 2015.

[19] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2015.

[20] "Developer guide - bitcoin." https://bitcoin.org/en/developer-guide. (Accessed on 06/12/2017).

[21] "Protocol documentation - bitcoin wiki." https://en.bitcoin.it/wiki/Protocol_documentation. (Accessed on 06/12/2017).

[22] "Hashed timelock contracts - bitcoin wiki." https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts, November 2016. (Accessed on 06/25/2017).

[23] R. Patel, "What are bitcoin payment service providers?." https://blog.cex.io/bitcoin-dictionary/what-is-psp-10856, October 2014. (Accessed on 06/25/2017).